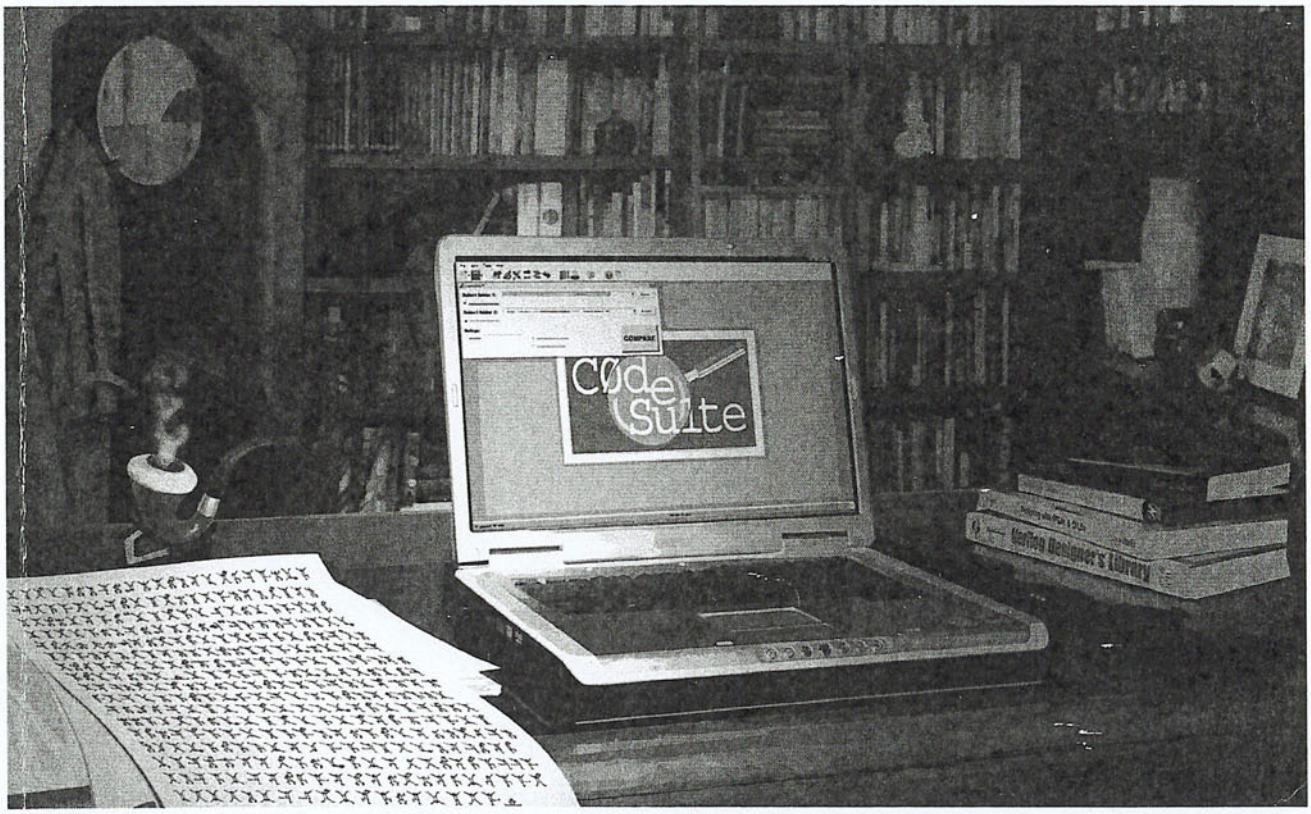
Exhibit A

PRENTICE HALL

THE SOFTWARE IP DETECTIVE'S HANDBOOK

Measurement, Comparison, and Infringement Detection



BOB ZEIDMAN

THE SOFTWARE IP DETECTIVE'S HANDBOOK

The Software IP Detective's Handbook

Measurement, Comparison, and Infringement Detection

Robert Zeidman
Software Analysis and Forensic Engineering Corporation

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data is on file with the Library of Congress

ISBN-13: 978-0-13-703533-5
ISBN-10: 0-13-703533-0

This product is printed digitally on demand.

First printing, April 2011

Editor-in-Chief
Mark L. Taub

Acquisitions Editor
Bernard Goodwin

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Barbara Wood

Indexer
Lenity Mauhar

Proofreader
Linda Begley

Cover Designer
Anne Jones

Compositor
LaurelTech

CONTENTS

16 IMPLEMENTATION

Now that I have discussed the theory behind source code correlation, I will discuss ways of practically and efficiently implementing that theory.[1] The particular implementation discussed in this chapter is based on the implementation used in the commercial CodeMatch tool that is a function of the CodeSuite program available from S.A.F.E. Corporation. This tool focuses on finding software copyright infringement, so many of the implementation choices are based on optimizing that use of source code correlation.

The implementation I describe here makes use of a basic knowledge of programming languages and program structures to simplify the task of comparing and identifying matching program elements. It is not necessary to implement a full programming language parser because the exact functionality of the statements does not need to be known (this will not be the case, however, when functional correlation is developed). The implementation must only recognize programming language keywords that are specific to the programming language of the program being examined. In addition, this implementation needs to know the characters that are used to delimit comments, the characters that are used to delimit strings, and the characters called "separators," which are the characters that cannot be used in identifier names. For example, operators (such as * / + −) are separators because when parsing a statement, reaching one of these means

1. Note that some of the concepts discussed in this chapter may be covered by patents that have been issued or are pending.

the end of an identifier has been reached (typically, though not always, whitespace is reached first). On the other hand, the underscore character (_) is valid within an identifier name in many programming languages, so it is not a separator.

The implementation needs to separate programs into five elements: statements, comments, strings, instructions, and identifiers. Note that these are not mutually exclusive; statements can include instructions, identifiers, and strings. It is useful to put these elements of code into arrays.

16.1 CREATING ARRAYS FROM SOURCE CODE

The first step in creating the arrays is to create two arrays for each source code file being examined. Consider the small snippet of C code shown in Listing 16.1.

Listing 16.1 Code Snippet 1

```
// ---- begin routine ----
void fdiv(
    char *fname,          // file name
    char *pathString)     // path
{
    int Index1, j;
    printf("Hello world!\n");

    if (strlen(pathString) == 0)
        strcpy(pathString, "C:\\Windows\\");
    else
        strcat(pathString, "\\");

    while (Index1 < 100)
    {
        printf("Getting results.\n");
        j = strlen(fname);
        j += strlen(PathString);
        /* find the file extension */
```

Table 16.1 shows how this small sample would lead to two arrays of statements and comments/strings. Note that whitespace has been reduced. This involves trimming all whitespace characters on the right and left of each string and

17 APPLICATIONS

There are many potential applications for source code correlation. In this chapter I discuss some of the ones that I have identified, though there are likely many more than these. Table 17.1 shows the applications that I discuss in this chapter and the most useful element correlations to emphasize in each case; each application is likely to emphasize certain element correlations over others, though these are not hard-and-fast rules.

Table 17.1 Applications of Source Code Correlation

Type of application	Emphasized correlations
Identifying authorship	Comment/string correlation Identifier correlation
Identifying origin	Comment/string correlation Identifier correlation
Detecting copyright infringement (plagiarism)	Statement correlation Comment/string correlation Identifier correlation Instruction sequence correlation
Detecting trade secret theft	Statement correlation Comment/string correlation Identifier correlation Instruction sequence correlation Functional correlation

Continues

Table 17.1 Applications of Source Code Correlation *(Continued)*

Type of application	Emphasized correlations
Locating third-party (open source) code	Statement correlation Comment/string correlation Identifier correlation Instruction sequence correlation
Compiler optimization	Statement correlation Instruction sequence correlation Functional correlation
Refactoring	Statement correlation Comment correlation Instruction sequence correlation Functional correlation
Detecting patent infringement	Functional correlation

17.1 FUNCTIONAL CORRELATION

Note that Table 17.1 mentions "functional correlation." This correlation involves functional elements of the code. I do not discuss the particulars of functional correlation in the book because it is a wide-open topic. Many techniques have been developed to define and analyze the functionality of programs. As of yet, I have not come across a tool that meets the following four criteria:

1. Is completely automatic
2. Is programming-language-independent
3. Completes in a reasonable amount of time
4. Produces scores that can fit into the source code correlation framework

The first criterion means that source code files are compared without human intervention. Once the tool is pointed at two sets of source code files, it calculates functional correlation without the need for the user to intervene. Certainly it is possible for a useful tool to calculate functional correlation and still require human intervention, but ideally such operator interference would not be necessary.

The second criterion means that two programs in two different programming languages could be compared. This is particularly important because functional

correlation should be comparing program functionality independently of the programming language. A starting point could be a tool that requires the programs being compared to be written in the same programming language. Such a tool could be useful but would be limited. Practical uses of functional correlation such as detecting trade secret theft and patent infringement would be most useful if one were able to compare programs written in different programming languages.

The third criterion has been a difficult goal to reach so far. Analyzing the functionality of a program seems to take exponential time with respect to the size of the program. Comparing two programs involves creating an abstract representation of the functionality of each program (something not well defined at this time and for which there is currently no standard) and comparing the two abstract representations. Many of the efforts to date attempt to create a representation of the entire program, which certainly makes a lot of sense. However, given the computational time constraints, perhaps it makes more sense to break the functionality down file by file and simply compare files as is done for the other correlation elements (statements, comments/strings, identifiers, and instruction sequences).

Once functionality can be automatically extracted from a program in a reasonable amount of time and independently of the programming language, the final criterion is to compare two functional representations in such a way that a normalized score is created. It is an open question how to accomplish this.

17.2 IDENTIFYING AUTHORSHIP

To use source code correlation to determine authorship of a program or an algorithm within a program, it would be most useful to find correlation between those elements that would be most likely to be unique to particular programmers. These elements are the comments, strings, and identifier names. Obviously any correlation used to determine authorship would emphasize comment/string correlation and identifier correlation.

There are instances when it is important to identify the author of source code. This could occur during legal disputes when two or more parties claim to have written a particular program. It could occur when two parties have jointly developed code for a program and, because of a contractual dispute or a royalty determination, must figure out which code was written by which party. While this may seem like an unusual scenario, I have been personally involved in several

cases where a large software manufacturer has asked me to search for its own programs online, in surplus software stores, and in my own private computer software collection because the company cannot find an earlier version of its software. This occurs more often than you might think in the software industry, sometimes because of the lack of strict procedures at many companies. It also happens because many modern software companies start very small and grow very large in a short period of time, and the emphasis is on getting products out faster than their larger competitors. The limited resources and short market windows mean that developing software quickly takes precedence over installing and using version control systems to keep track of software changes.

Determining authorship can also be useful in tax cases. Tax rates vary according to jurisdictions (for example, city, county, state, country). When software is partially developed overseas, for example, the IRS of the U.S. government can tax the foreign-developed code at a different rate from the domestically developed code. When multiple versions of code have been developed in multiple countries by multiple teams using various tools, the company may not have kept track of which code was developed where. Again, source code correlation can be used to help make this determination.

Identifying authorship can be useful to computer historians to determine whether certain programmers are responsible for writing certain programs or portions of programs. Historians use this kind of technique to determine authorship of works of literature and works of art. As computers become a part of history, computer historians may need to do something similar to correctly attribute code to its authors. Literary works are almost always published with the author's name on the work, yet there are still controversies and there are known cases of fraud. There are those who believe that William Shakespeare was the pseudonym for writer Christopher Marlowe, philosopher and scientist Francis Bacon, or Edward de Vere, the 17th Earl of Oxford and Lord Great Chamberlain of England. There have also been famous forgeries of artwork. One such example is three Etruscan terra-cotta warriors. They were purchased in the years 1915 through 1918 and put on display in 1933 as examples of the work of the ancient Etruscans, a people that lived in Italy around 800 BCE. It was not until 1960 that chemical tests determined that modern materials were used and that, in fact, the pieces were created at the time of their sale by Italian brothers Pio and Alfonso Riccardi and their sons.

While it seems unlikely that source code would ever be as valuable as a historical artifact to warrant the attention of forgers, historians have already begun collecting, studying, and storing computer hardware and software in places like

the Computer History Museum in Mountain View, California. It is conceivable that these historians may want to determine or even locate the programmer for a particular piece of code, and source code correlation can help.

Another use for identifying authorship is for tracking viruses, spyware, and other kinds of cyber-security threats. Identifying the origin of software, as described in the next section, as well as authorship is useful in this important field. Comparing source code for a computer virus against source code from known programmers, particularly those with a history of such illicit activity, can help catch these criminals and convict them at trial.

17.3 IDENTIFYING ORIGIN

As with determining authorship of a program or an algorithm within a program, identifying origin focuses on the correlation between those elements that would most likely be unique to programmers in particular companies or countries. These elements are the comments, strings, and identifier names. Any correlation used to determine origin would emphasize comment/string correlation and identifier correlation.

Identifying the origin of a program or a piece of a program can be useful for the same reasons as identifying authorship is useful. Because many modern programs are written by teams of engineers, it may not be possible to identify the actual author, but it may be possible to identify the company where the software was developed. Different companies have different coding standards and different corporate cultures that could result in higher correlation between programs developed within the company than between programs developed at different companies.

Similarly, it may be possible to identify the country of origin of a program. In particular, not only might coding standards and coding styles be similar among programmers in a particular country, but there would obviously be language differences in the identifier names and particularly in the comments and strings. Even in countries where programmers use the same language, idioms, colloquialisms, and slang would differentiate countries or areas within countries where the software was developed.

Identifying the country of origin can be useful in legal disputes, as described previously. Country of origin may be even more useful in tracing software intended to infiltrate another country's in an act of cyber warfare, a growing and dangerous threat in the modern world that can potentially bring down an entire nation without a single missile or weapon of mass destruction.

17.4 DETECTING COPYRIGHT INFRINGEMENT (PLAGIARISM)

The most common use of source code correlation to date is to find copyright infringement, which is essentially the legal term for plagiarism. Source code correlation was initially designed to help detect copyright infringement, and it has been successfully used in over 50 cases at the time of this writing.

To find copyright infringement, it is necessary to find all signs of copying, especially considering that modifications may have been made subsequent to the unauthorized copying. The modifications may be explained by the normal development process, including debugging and feature enhancements and additions. The modifications may also be the result of deliberate attempts to hide the copying. For these reasons, source code correlation used to detect copyright infringement must emphasize correlation of all literal elements: statements, comments and strings, identifiers, and instruction sequences.

17.5 DETECTING TRADE SECRET THEFT

Another common use of source code correlation is the detection of trade secret theft. When literal elements of a program's source code are copied, not only does that constitute copyright infringement, but it is also often a case of trade secret theft. This is because much software source code, other than open source software, meets the requirements of being a trade secret that were discussed in Chapter 8:

1. It is not generally known to the public. Programmers may know how to implement particular functions using source code, but each program has many unique functions that are implemented in ways that are not generally known because they are specific to the particular program.
2. It confers some sort of economic benefit on its holder, where the benefit is from not being known to the public. Most software is difficult to write, taking years of education and experience to do correctly. Most code has an economic benefit to its owner because of the knowledge and effort that went into creating it. Also, it can be argued that understanding source code allows a competitor to learn a program's weaknesses that can be exploited, lowering the program's value.
3. The owner of the trade secret makes reasonable efforts to maintain its secrecy. Software source code is almost always kept secret.

Consequently, in the case of literal copying, not only is source code correlation useful for detecting copyright infringement, but it is also useful for finding

trade secret theft using all four algorithms—statement correlation, comment/string correlation, identifier correlation, and instruction sequence correlation.

However, trade secret theft can also involve copying a method for accomplishing some function without copying the actual source code that implements that function. For this reason, source code correlation that would find all forms of trade secret theft would include functional correlation. As of today, there is not an efficient method for determining functional correlation in a reasonable amount of time. A tool for computing a correlation that can accurately detect trade secret theft is still in the future, as of this writing.

17.6 Locating Third-Party Code (Open Source)

As open source code is finding its way into more and more products, it is becoming a more complex issue. Open source code is covered by different licenses, the most common one being the GNU Public License (GPL) that has certain strict requirements, particularly about making source code available to the public. In many cases, open source code has crept into software projects without the managers or many of the programmers being aware. Sometimes open source code has been used in a project even though the project owners do not intend to abide by the licensing terms. Companies are now getting nervous about this phenomenon, especially because several successful lawsuits have been brought by the open source community of developers against commercial companies. Many companies now employ sophisticated means of determining whether their own software products contain open source code, and source code correlation can be used to help make that determination.

The issues associated with detecting open source code are really a subset of the issues associated with finding any third-party source code in a project. There may be instances of commercial third-party code for which the license has expired or that was obtained by unlawful means. Source code correlation can be a useful tool for detecting the use of such third-party code, whether it is open source or commercial.

In most cases, the third-party code has not been modified, or has not been modified significantly, because typically the source code is used unintentionally and so there is no attempt to obscure its use in the program. Even when the source code is used with the knowledge that it is not lawful, the users do not expect it to be uncovered because they are not releasing their source code (see Part VI). Also, third-party code is often not modified, or modified only slightly, because an advantage of third-party code is that it has been written

and vigorously tested, and modifying the code would require further testing and debugging. In these cases, source code correlation may be overkill, and techniques that look for exact matches, like source code differentiation (see Part IV) or comparing hashes of the source code, may be sufficient.

However, if an attempt has been made to hide the third-party code, or it is believed that such an attempt was made, then source code correlation used to detect third-party code must emphasize the same elements as those used for detecting plagiarism. This involves all literal elements: statements, comments and strings, identifiers, and instruction sequences.

17.7 COMPILER OPTIMIZATION

It is possible that source code correlation can be used to eliminate redundancy in code during the compilation process, thereby optimizing the compilation. In this case, it seems that the correlation should emphasize statement correlation and instruction sequence correlation and, of course, functional correlation when that becomes available. By using source code correlation, a compiler could find highly correlated sections of code. Once a compilation has been completed for one of these sections of code, the compiler may be able to speed up the effort on highly correlated sections of code without needing to perform another full, time-consuming optimization process.

17.8 REFACTORING

An area of growing interest in computer science, and in industry, is refactoring. This is the process of cleaning up and simplifying software source code to make it more readable and maintainable, without changing its functionality. According to Martin Fowler, one of the early gurus of refactoring, "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

One particular aspect of refactoring is clone detection and elimination. A clone is a copy of a section of code that performs a function that is identical to that of the original code. In an early paper on clone detection entitled "Using Clone Detection to Manage a Product Line," Ira Baxter and Dale Churchett describe clone detection and its benefits as follows:

> Clone detection finds code in large software systems that has been replicated and modified by hand. Remarkably, clone detection works because people copy conceptually identifiable blocks of code, and make only a few changes, which means the same syntax is detectably repeated. . . . Clones can thus enhance a product line development in a number of ways: removal of redundant code, lowering maintenance costs, identification of domain concepts for use in the present system or the next, and identification of parameterized reusable implementations.

Obviously it is unnecessary and inefficient to have cloned code within a single product because it just increases the code size and increases the maintenance effort without providing significant additional functionality. Eliminating clones is a useful exercise if it can be done efficiently, and source code correlation can be an efficient method of doing so. Correlation used for clone detection and elimination would emphasize the following elements of source code correlation: statement correlation, comment correlation, instruction sequence correlation, and functional correlation.

Notice that I refer to comment correlation rather than comment/string correlation. It might not seem obvious at first to include comment correlation. Here we are concerned mostly about the descriptions provided in the comments, and this may be one area where comments and strings should be treated differently rather than combined into a single correlation score. Comments, particularly headers that describe the functionality of a routine in the program, could provide important details about the correlation of two functions in the program. For example, it is possible that two functions, written by two different programmers, would differ significantly in their implementation but in fact implement the same function. Depending on the efficiency, and availability, of a functional correlation measure, comment correlation could be a way of determining that two or more functions perform the same operation and can be replaced with a single function, even though their other correlation scores are low.

17.9 DETECTING PATENT INFRINGEMENT

Patent infringement essentially involves the creation of a program that uses a patented method by someone other than the person or entity that owns the patent and has not been given that right by the patent owner. Patent infringement occurs even when the infringer has never seen the source code of the patented invention. For this reason, calculating the correlation of the literal

elements of a program is generally not useful unless source code was also copied. For the vast majority of patent cases, source code correlation would emphasize functional correlation, which, as explained above, is not yet available. When an efficient, accurate method of functional correlation is found, there will be many uses for it.